ISI Research Report
ISI/RR-89-227
July 1989

(4)

Walid Najjar
Jean-Luc Jezouin
Jean-Luc Gaudiot

Parallelism in the Discrete-Event Simulation Algorithm

INFORMATION SCIENCES INSTITUTE

213/822-1511

4676 Admiralty Way/Marina del Rey/California 90292-6695

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT This document is approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ISI/RR-89-227	5. MONITORING ORGANIZATION REPORT NUMBER(S) -------
6a. NAME OF PERFORMING ORGANIZATION USC/Information Sciences Institute 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 4676 Admiralty Way Marina del Rey, CA 90292-6695	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy Street Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SDIO NSF 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-87-K-0022 CCR-8603772
8c. ADDRESS (City, State, and ZIP Code) Strategic Defense Initiative Organization Office of the Secretary of Defense (over) The Pentagon, Washington, DC 20301	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. ------- / PROJECT NO. ------- / TASK NO. ------- / WORK UNIT ACCESSION NO. -------

11 TITLE (Include Security Classification)

Parallelism in the Discrete-Event Simulation Algorithm (Unclassified)

12. PERSONAL AUTHOR(S) Najjar, Walid; Jezouin, Jean-Luc; Gaudiot, Jean-Luc

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Research Report	FROM _____ TO _____	1989, July	16

16. SUPPLEMENTARY NOTATION

Rassul Ayani of the Royal Institute of Technology, Sweden, recently wrote his doctoral thesis on the scheme proposed in this paper and improved on it. Interested readers can contact him directly.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

discrete-event simulation, parallelism

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

With the increasing complexity of VLSI circuits, simulation on a digital computer system has become a primary means of low-cost testing of new designs. However, a detailed behavioral simulation can be highly expensive in terms of computation loads. Simulation time can be reduced by *distributing* the problem over several processors. Indeed, the availability of commercial multiprocessors gives a new importance to parallel and distributed simulation. This report surveys the techniques that have been proposed to deal with this problem and then presents a new scheme based on a consistent description of the system to be simulated, which allows a maximum exploitation of the parallelism inherent in the system.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Victor Brown Sheila Coyazo	22b. TELEPHONE (Include Area Code) 213/822-1511 / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

8c. (continued)

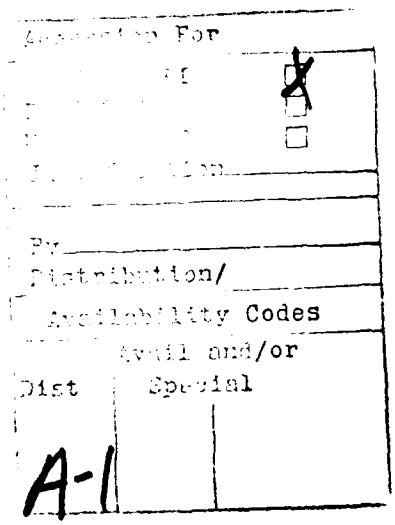
National Science Foundation
1800 G Street NW
Washington, DC 20550

ISI Research Report
ISI/RR-89-227
July 1989

University of Southern California

Walid Najjar
Jean-Luc Jezouin
Jean-Luc Gaudiot

Parallelism in the Discrete-Event Simulation Algorithm



INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

This research is supported by the Strategic Defense Initiative Office under Office of Naval Research Contract No. N00014-87-K-0022, and by the National Science Foundation under Grant No. CCR-8603772. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of SDIO, ONR, NSF, or the U. S. Government. No official Government endorsement should be inferred.

1 Introduction[1]

The design and prototyping of more complex and powerful computer systems relies heavily on testing of new designs by simulation. Furthermore, the relatively low-cost availability of computing equipment has introduced simulation into many diverse areas such as agriculture, biology, and econometrics, in addition to engineering and scientific research. The advent of operational parallel and distributed architectures has increased the interest in distributing the execution of simulation programs over several processors in order to reduce their execution time.

Traditionally, simulation algorithms have been identified as either *time-driven* or *event-driven*. The time-driven model reflects all the variations in the system being modeled, provided a sufficiently low time granularity. It is very efficient in modeling continuous change; but its use is impractical however in modeling systems where change is by discrete steps. Event-driven simulation, on the other hand, is efficient in modeling the asynchronous occurrence of *discrete* events in time. For such systems, it provides a good modeling accuracy and a higher execution efficiency.

However, the discrete-event simulation algorithm is essentially a highly sequential algorithm that relies on the centralized notion of an *event queue* and a *simulation time*. Distributing these over several processors can imply a large overhead to ensure the *sequential consistency* of the simulation. A distributed simulation is sequentially consistent when events occur in the same order as in a sequential version.

The scheme proposed in this report, while maintaining a central event queue and a common simulation time, allows the concurrent execution of events over several processors and guarantees a sequentially consistent simulation. The possible concurrency among events is detected at compile time, and the run-time overhead is kept to a minimum.

2 Simulation Models

2.1 Event- and Time-Driven Simulation

The time- and event-driven simulations are essentially equivalent algorithms. Both have the same modeling power. The main difference between them is in their respective expected performance for a given problem.

[1]Part of the work described in this report has appeared in [1] and [2].

In the *time-driven* model, the simulation time is moved up by a constant amount at each iteration of the simulation algorithm. At each step, the simulator proceeds to the "next instant." A generic algorithm for time-driven simulation is as follows:

repeat
$t = t + \Delta t$
foreach element in the system **do**
evaluate new state
post global state changes
until (End-of-Simulation)

All elements of the system are evaluated at each iteration, regardless of their activity status. This approach is highly efficient if, on the average, a large fraction of the system elements are active during any simulation time interval, and every time interval witnesses changes in the state of the system. Provided a sufficiently small time interval, this approach can model, with high fidelity, systems with time-continuous variables (such as electric voltages in circuit simulation).

In the *event-driven* simulation model, only the values of elements that have actually changed are updated. The simulation time is moved up at each step to the "next event" time. All "future" events are maintained in a *simulation time* ordered list. A generic event-driven algorithm can be described as follows:

repeat
t_{next} = time(next-event)
foreach event posted at t_{next}
evaluate
schedule any new events generated
until (End-of-Simulation)

All events scheduled at the current simulation time are retrieved and evaluated; newly generated events are scheduled on the event list, and the simulation time is advanced to the time of the next scheduled event. This approach is particularly suited for modeling discrete systems where state changes occur in discrete increments.

2.2 Concurrency in Simulation Models

In essence, a simulator runs an algorithmic description of a system. The concurrency delivered by the execution of this algorithm is several fold:

1. *Element concurrency* exists when several elements of the system can be evaluated at the same time. It is specific to the time-driven model, where all elements are evaluated at every time step. For instance, when several gates of a complex logic circuit receive all their inputs from the same outside elements, the state of these gates can always be evaluated concurrently without a risk of dependency. An example of such an implementation is found in the IBM Yorktown Simulation Engine [3,4,5].

2. *Time concurrency* results from the simultaneous occurrence of changes within the system. In other words, it exists when several unrelated events are scheduled to happen at the same simulation time, in an event-driven environment. Such a feature can be found in both the Daisy [6] and ZYCAD [7] machines.

3. *Control concurrency* consists of executing (in a pipelined fashion) the tasks that are at the core of the event-driven model: Retrieve, Evaluate, and Schedule events. It has been described in [8] and implemented in [6].

4. *Object concurrency* means that a set of logically related activities can be grouped in an *object*. Often these objects exhibit a low degree of interaction and thus can be evaluated in parallel with little synchronization overhead.

2.3 Distributed Simulation

The aim of distributed simulation is to map the simulation model over several loosely coupled processors, where objects execute locally and exchange information in the form of *time-stamped messages*. Two main paradigms have been proposed that implement distributed discrete-event simulation: the *Network Paradigm* [9,10] and the *Time-Warp Mechanism* [11,12]; both are asynchronous algorithms.

The Network Paradigm. This model was proposed independently by Peacock et al. [9,13] and Chandy et al. [14,10]. It can be described as a *conservative asynchronous mechanism*. It is *conservative* because it assumes that synchronization between any two events may be needed until proof that it is not required.

Simulation is modeled as a directed graph, where arcs represent messages passing among objects and carry a monotonic, non-decreasing, simulation-

time- ordered sequence of events. Every node has a local simulation time, otherwise called *next event time*. Each input link in a node corresponds a *link time*, which is the value of the time stamp of the last message received on that link. The next event in a node is chosen as the *minimum link time* event.

This method suffers from the possible introduction of deadlock situations. Several schemes have been proposed as a remedy to this problem:

- The Link Time Algorithm [9] reduces the probability of a deadlock.
- The Blocking Table Algorithm [13] allows the distributed detection of deadlocks, however, with a time complexity $O(n^3)$.
- The Controller Method [14] relies on a central controller that runs a deadlock detection algorithm and initiates recovery, at the expense of a potential bottleneck.

The Time-Warp Mechanism. This mechanism, proposed by Jefferson and Sowizral [11], is an *asynchronous optimistic mechanism.* It is *optimistic* because it assumes (hopes for) independency among events, and implements a *rollback and undo* when these conditions are not verified. Essentially, it relaxes the condition of monotonic, non-decreasing, time-stamped messages along arcs, allowing out-of-order arrival of events between two logic processes. This method suffers from two main drawbacks: a potential for domino effect in rollbacks, and a large space overhead necessary to maintain the past history of each process.

Gafni [12] proposed a scheme that reduces the amount of space overhead by implementing a garbage collection mechanism. Lavenberg et al. [15] present an analytical evaluation of such a mechanism for two processes mapped onto two processors. Their results show that a good speed-up can be obtained when the probability of interaction among processes is low (0.05 or less). The performance degrades for larger probabilities.

3 Detection of Parallelism

The discrete-event simulation algorithm is essentially sequential. The Network Paradigm and the Time-Warp Mechanism aim at speeding up the simulation by distributing the central event queue over a network of processors. The methodology described in this report aims at detecting the possible concurrency among events in a central event queue and thereby exploiting any potential parallelism.

3.1 System Modeling

Let Σ be the model of a system under simulation. It can be decomposed into a set of *independent subsystems*, or *disjoint objects.*[1] Σ can be seen as the set of *state variables* describing a system and each σ_i as a subset of Σ.

$$\begin{cases} \forall i \neq j, \sigma_i \cap \sigma_j = \emptyset \\ \bigcup_{i=1}^{n} \sigma_i = \Sigma \quad (n \geq 1) \end{cases} \tag{1}$$

Equation 1 states that the partitions of Σ are non-overlapping. Let $S(\Sigma)$ be the set of all possible states of Σ, and, for each subsystem (object) σ_i, let $S(\sigma_i)$ be the set of all possible states of σ_i. Then we can express the above partitioning as a concatenation of states

$$S(\Sigma) \; = \; (S(\sigma_1), S(\sigma_2), ..., S(\sigma_n))$$

An *event* in the system can therefore be defined as a state transition over some subset σ_i of Σ occurring at time t. An event in σ_i at time t_j can be described as

$$e(\sigma_i, t_j) \equiv \{S(\sigma_i) = s_1 \;\; t < t_j\} \wedge \{S(\sigma_i) = s_2 \;\; t > t_j\} \quad s_1 \neq s_2 \tag{2}$$

The evaluation of an event can result in the creation of other events within the same subsystem or in other subsystems. These events are said to be **induced** by the event that was evaluated. The state of a subsystem can therefore be affected by events in one or more other subsystems. A relation of causality or *functional dependence*, between any two subsystems can be defined by

Definition 1 *A subsystem σ_i is* **functionally dependent** *upon a subsystem σ_j iff some event $e(\sigma_i, t)$ can* **induce** *an event $e'(\sigma_j, t')$ in* **finite time.**

This relation is denoted by $\sigma_i \Rightarrow \sigma_j$. From this definition we can model a system with an *Object-Dependency Graph*, which is a directed graph $G = (V, E)$, where

$$V = \{\sigma \subset \Sigma\}$$

is the set of nodes, and

$$E = \{(\sigma_i, \sigma_j) \mid \sigma_i \Rightarrow \sigma_j \;\; \sigma_i \subset \Sigma, \; \sigma_j \subset \Sigma\}$$

[1] In this report the terms *subsystem* and *object* will be used interchangeably.

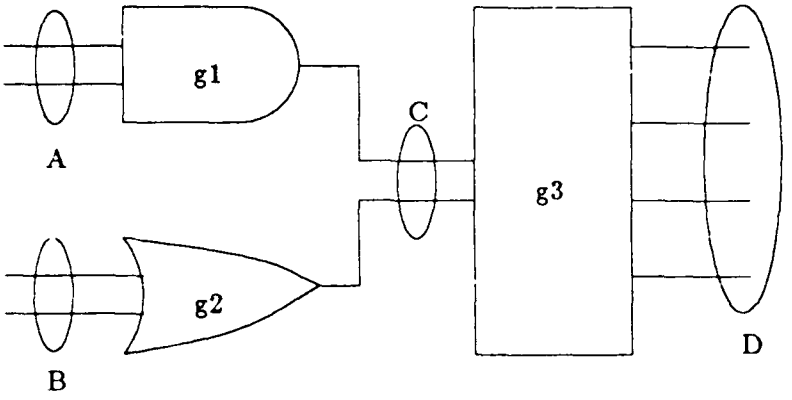


Figure 1: Modeling a simple digital circuit.

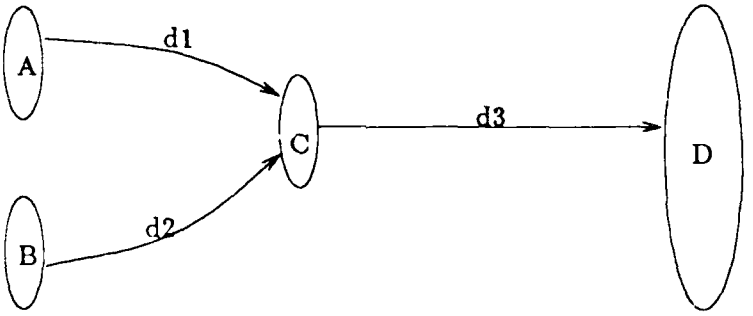


Figure 2: Object-dependency graph.

is the set of edges.

Figures 1 and 2 show the example of a simple digital circuit, its partitioning into a set of objects A, B, C, and D, and the corresponding object dependency graph.

Based on the relation of functional dependence, we can define a *directional distance* between two objects as

Definition 2 $\delta(\sigma_i, \sigma_j)$ = *minimum possible delay between any event* $e(\sigma_i, t)$ *and an induced event* $e'(\sigma_j, t')$.

In other words, $\delta(\sigma_i, \sigma_j)$ is the lower bound on the possible delay between any event in σ_i and its possible effect in σ_j. If $\delta(\sigma_i, \sigma_j) = +\infty$ then *no* event in σ can induce an event in σ_j.

In Figure 1, the directional distances $d1$, $d2$, and $d3$ correspond to the respective delays in gates 1, 2, and 3.

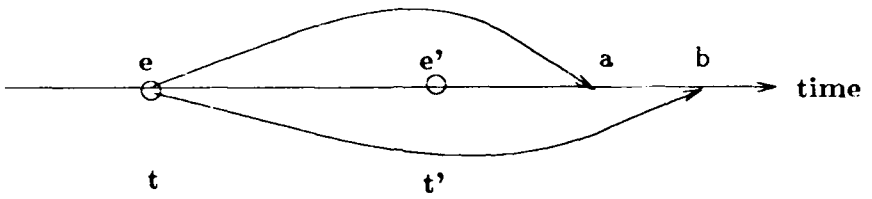


Figure 3: Possible event concurrency.

3.2 Scheduling Relations

Two subsystems are totally independent when

$$\delta(\sigma, \sigma') = \delta(\sigma', \sigma) = +\infty$$

In other words, neither one can influence the other. Therefore, any event occurring in one of these objects cannot influence the evaluation of any event in the other. Such two events can obviously be evaluated concurrently. This remark enables us to define a relation of *strict independence* ($\Im$) between two events:

$$e(\sigma, t) \ \Im \ e(\sigma', t') \equiv \neg(\sigma \Rightarrow \sigma') \wedge \neg(\sigma' \Rightarrow \sigma)$$

This means that two events are *strictly independent* if and only if the respective subsystems to which they belong are functionally independent. Verifying this relation reduces to finding the *non-connected components* of the model graph G. Even though all the concurrency among events will be detected *across* independent subsystems, this relation will miss most of the parallelism resulting from *pipelining of events* along the same (possibly cyclical) path of the graph.

Indeed, when two events e and e' are generated inside two subsystems σ and σ' such that $\sigma \Rightarrow \sigma'$, their times of occurrence can be compatible with a concurrent evaluation.

Figure 3 shows an event history with $e(\sigma, t)$ and $e'(\sigma', t')$ where $(t' - t) < \delta(\sigma, \sigma')$. In this case, all events (e.g., a and b) that can possibly be induced by the evaluation of e will fall in the future of e' and cannot affect its evaluation. Therefore, events e and e' could be evaluated concurrently without affecting the correctness of the simulation.

In general, $e(\sigma, t)$ and $e'(\sigma', t')$ can be evaluated concurrently if and only if the result of evaluating e have consequences, i.e., induces events, in σ'

later than t'. Since $\delta(\sigma, \sigma')$ is defined as the *minimum delay* between the two subsystems, the following inequality must hold for the concurrent evaluation to be correct:

$$(t' - t) < \delta(\sigma, \sigma')$$

Because of a possible cycle in the dependency graph, the reciprocal must also hold. We define, therefore, a relation of *generalized independence* ($\Re$) between two events (object concurrency):

$$e(\sigma, t) \ \Re \ e'(\sigma', t') \equiv [t - t' < \delta(\sigma', \sigma)] \wedge [t' - t < \delta(\sigma, \sigma')]$$

Therefore, determining the possible concurrency of two events amounts to comparing the interval of time between their respective occurrences to the time distance δ between the two subsystems to which they belong. $\Re$ is not an equivalence relation since it is not transitive. This implies that deciding whether several events can be evaluated concurrently at a given time requires the examination all pairs of candidates.

The concurrency among events is not limited to events that are time consecutive. At any simulation time t_{sim}, the content of the event queue can be described by a time-ordered sequence of events, where the i^{th} event at simulation time t_i is denoted by $e_i(t_i)$:

$$Q(t_{sim}) = (e_1(t_1), e_2(t_2), ..., e_i(t_i), ..., e_n(t_n)) \quad t_{sim} \leq t_i \leq t_j \ \ i < j$$

In the sequence $Q(t_{sim})$, the set of independent (and therefore concurrent) events is defined by

$$C(t_{sim}) = \{e_i(t_i) \mid \forall j < i \ \ e_i \ \Re \ e_j\} \tag{3}$$

The definition of the set $C(t_{sim})$ states that any event with an associated simulation time $t_i > t_{sim}$ can be evaluated at simulation time t_{sim} iff this event is independent of *all* the events preceding it in the sequence $Q(t_{sim})$. Obviously, the first element in the $Q(t_{sim})$ sequence is also the first element in $C(t_{sim})$. Note that, once an event is in $C(t_{sim})$, it stays a member of the set until it is evaluated. In other words,

$$e(t) \in C(t_0), \ \Rightarrow e(t) \in C(t_1) \ \ \forall t_0 \leq t_1 \leq t$$

Therefore, $C(t_0)$ constitutes the set of independent events that can be evaluated concurrently at any simulation time $t \geq t_0$.

3.3 Implementation Considerations

In this section we describe a scheme for the run-time detection of concurrency among events and the parallel execution of these events. The implementation of such a scheme relies on building a *Delay Table* D, which is the description of the object-dependency graph for the system under simulation. D is an $n \times n$ array, where n is the number of objects. The delay values are defined as follows:

$$D[i,j] = \begin{cases} \delta(\sigma_i, \sigma_j) & i \neq j \\ 0 & i = j \end{cases} \qquad (4)$$

The diagonal of this matrix is null, since there is no delay between a state transition in a subsystem and its effects on that subsystem. This prevents two events pertaining to the same subsystem from being evaluated concurrently. If the object-dependency graph is acyclic, D is an upper triangular matrix, the values in the lower half being equal to infinity.

The Delay Table is created at compile time by an analysis of the object-dependency graph, which yields the lower bounds on the delays between dependent objects in the system. The set of concurrent events at simulation time t can be built at run-time from the Delay Table and the event queue.

On a multiprocessor with m processors, one is dedicated to running the event queue management tasks, allowing up to $m-1$ events to be evaluated concurrently. Hence the event-driven simulation algorithm becomes

for every simulation time t;
repeat
  determine the set of independent events $C(t)$;
  schedule the execution of up to $(m-1)$ events from $C(t)$;
  update the simulation time to that of the new head of the queue;
  insert any new events in the queue;
until (End-of-Simulation)

Note that this algorithm guarantees a progress rate of the simulation at [illegible] equal to that of a sequential simulation. The head of the queue is [illegible] a member of $C(t)$ and is evaluated at every iteration. Therefore, the [illegible]ation time is always updated to at least that of the next event, as in the [illegible]ntial algorithm.

4 Conclusions

We have examined in this report the issue of simulation in a parallel environment. Although the problem is apparently centered around the centralized notion of time, several types of concurrency can be found in simulation models. Event-driven simulation has been shown to provide time, object, or control concurrency.

Using a formal description of a system under simulation, we were able to define relations of functional dependence among objects in the system, as well as two relations of strict and generalized independence between events. A compilation strategy has been described, based on these relations, that allows the run-time detection of parallelism in the event-driven simulation model. This strategy has been shown to preserve sequential consistency among events.

Directions for future research include evaluation of the distribution of the degree of parallelism that can be obtained using this method in applications such as switch-level simulation of logic circuits and the stochastic simulation of network of queues. Another direction is the implementation of a parallel simulation environment on a shared memory multiprocessor.

References

[1] W. Najjar, J-L. Jezouin, and J-L. Gaudiot. Parallel execution of discrete-event simulation. In *Proceedings of the 1987 International Conference on Computer Design*, October 1987.

[2] W. Najjar, J-L. Jezouin, and J-L. Gaudiot. Parallel discrete-event simulation on multiprocessors. *IEEE Design and Test*, 4(6):41–44, December 1987.

[3] G. Kronstadt and G. Pfister. Software support for the Yorktown Simulation Engine. In 19^{th} *Design Automation Conference*, pages 60–64, 1982.

[4] G. Pfister. The Yorktown simulation engine: Introduction. In 19^{th} *Design Automation Conference*, pages 51–54, 1982.

[5] M.R. Denneau. The Yorktown simulation engine. In 19^{th} *Design Automation Conference*, pages 55–59, 1982.

[6] W.G. Paseman and G. Catlin. *Hardware Acceleration of Logic Simulation Using a Data Flow Architecture.* Technical Report, Daisy System Corporation, 1984.

[7] Zycad LE-001 and LE-002 Logic Evaluator - Product description. ZYCAD Corporation, 1982.

[8] M. Abramovici, Y.H. Levendel, and P.R. Menon. A logic simulation machine. In 19^{th} *Design Automation Conference*, pages 65–73, March 1982.

[9] J.K. Peacock, W. Wong, and E. Manning. A distributed approach to queueing network simulation. In *Proceedings, IEEE 1979 Winter Simulation Conference*, pages 399–406, 1979.

[10] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

[11] D. Jefferson and H. Sowizral. Fast concurrent simulation using the time-warp mechanism. In *Proceedings, SCS Distributed Simulation Conference*, January 1985.

[12] A. Gafni. *Space Management and Cancellation for Time Warp.* PhD thesis, University of Southern California, December 1985.

[13] J.K. Peacock, W. Wong, and E. Manning. Distributed simulation using a network of processors. *Computer Networks*, 44–56, February 1979.

[14] K.M. Chandy, J. Misra, and V. Holmes. Distributed simulation of networks. *Computer Networks*, 3:105–113, 1979.

[15] S. Lavenberg, R. Muntz, and B. Samadi. Performance analysis of a rollback method for distributed simulation. In 9^{th} *International Symposium on Computer Performance Modeling, Measurement, and Evaluation*, pages 117–132, May 1983.